Squeeze More Code Into Your PSoC

Cypress Semiconductor White Paper

By : by Clyde Stubbs

CEO and Founder, HI-TECH Software

Executive Summary

New compiler technology effectively doubles the Flash on your existing PSoC by achieving nearly twice the code density of existing compilers. By optimizing pointers, registers and stacks, it can also free up 10% to 15% of SRAM resources. With "omniscient code generation" (OCG) technology, you can keep growing that application in the PSoC device you are currently using.

This white paper introduces the HI_TECH compiler and demonstrates its effectiveness with the Cypress' Programmable System on Chip (PSoC).

Introduction

The Cypress' Programmable System on Chip (PSoC) mixed signal array is a complete system level solution with configurable digital and analog peripherals, an 8-bit microcontroller, and embedded Flash and SRAM memory. The PSoC is an exceptionally flexible and cost effective solution. However, as applications grow, designers can stretch a mixed signal array to the limits of their SRAM and Flash densities. This document discusses how to decrease code size, in some cases dramatically so, by using HI-TECH Software's HI-TECH C PRO for the PSoC Mixed-Signal Array, a new compiler with Omniscient Code Generation technology that can double the density of existing C-code.

Until now, the only solutions when bumping up against SRAM and Flash memory limitations have been to:

- Limit the functionality of the end-product,
- Migrate the application to a larger PSoC device with more SRAM and Flash memory, or
- Hand craft assembly language code to reduce the program, stack and variable sizes -- an exceptionally cumbersome and time-consuming task that restricts the portability of the program code.

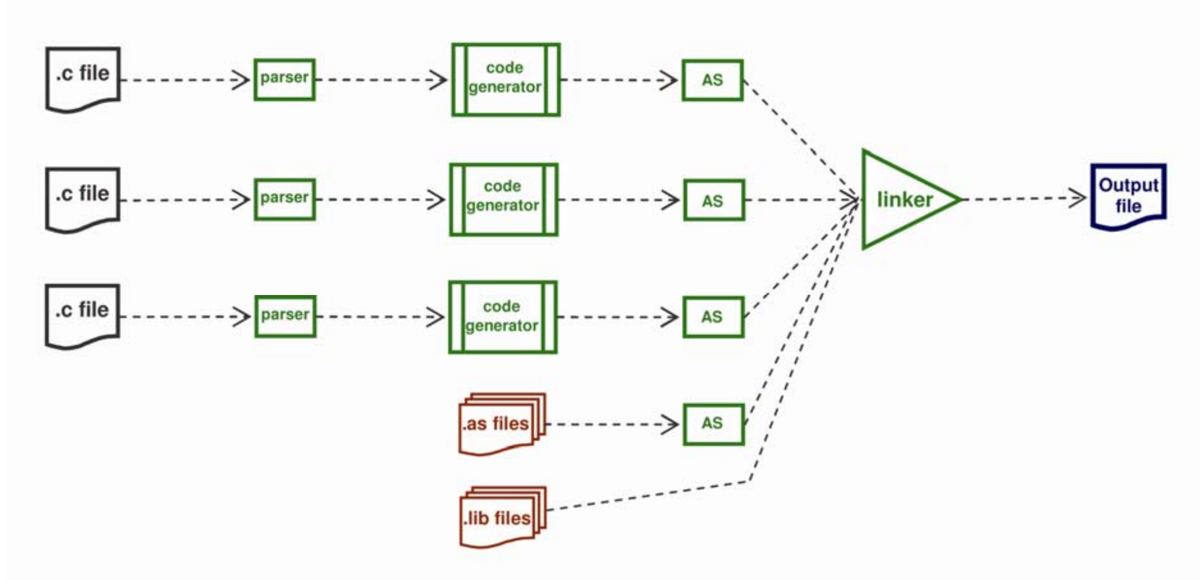
None of these alternatives is very attractive.

New compiler technology effectively doubles the Flash on your existing PSoC by achieving nearly twice the code density of existing compilers. By optimizing pointers, registers and stacks, it can also free up 10% to 15% of SRAM resources. With "omniscient code generation" (OCG) technology, you can keep growing that application in the PSoC device you are currently using.

Pitfalls of Existing Compilation Technology

Conventional compilation technology shadows the modular software design process. Typically programs are broken into modules, partly to accommodate the increased complexity of programs and partly to distribute programming tasks among teams of engineers to speed up the process. Compilers track this process, individually compiling each module into an independent sequence of low level machine instructions. Once all the modules are compiled, a linker links the modules together, along with any code being used from precompiled libraries.

Figure 1 Independent Compilation



The compiler never has complete information about the program being compiled. Although many compiler vendors claim “global optimization,” the optimization is done only within single modules. There is no optimization across all program modules, which leads to the sub-optimal allocation of stacks, registers, and memories. Multiple instances of the same routine may be unnecessarily repeated in different program modules. Reentrant code may exist in interrupt routines and in the main line program. Declarations of the same variable or object may not be consistent between modules. The result is code that takes up more of the PSoC’s flash memory than is necessary, and pointers and stacks that use too much SRAM.

Almost invariably it is necessary to make use of non-standard compiler features to hand craft the program to the target device architecture, compromising code portability.

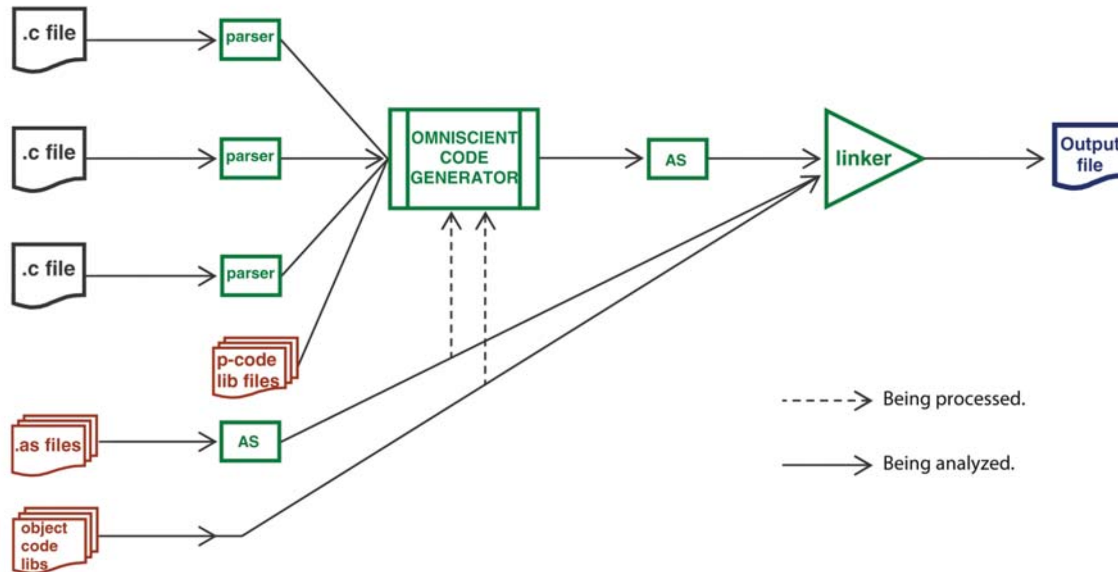
Until recently, it just has not occurred to compiler vendors that, aside from current software development methodologies, there is no reason to compile each program module independently. In fact, the reverse is true. By combining program modules into one large program, the compiler has the opportunity to identify and consolidate reentrant code, identify inconsistent variable declarations between modules, eliminate redundant code that may exist in more that one module and optimize all the stacks, registers and memories to exploit the unique advantages of the PSoC architecture, without handcrafting any code.

[+] Feedback

HI-TECH C PRO for the PSoC Mixed-Signal Array To the Rescue

A new compiler technology, called Omniscient Code Generation (OCG), analyses every variable, function, stack, register and pointer in each and every program module, *before* it generates the object code. OCG technology has been implemented in HI-TECH Software's new compiler, C PRO for the PSOC Mixed-Signal Array, available now.

Figure 2 – Compilation Using OCG



Rather than relying completely on the linker to uncover errors between the independently compiled modules, an OCG compiler defers object code generation until a view of the whole program is available. It does this by completing the initial stages of compilation for each module separately. Then, instead of compiling to machine code instructions in an object file, it compiles each module to an intermediate code file that represents a more abstract view of each module. Before producing any actual machine instructions or allocating any registers, it applies optimization algorithms across all program modules based on this global view of the program.

This approach solves most of the problems associated with conventional compilation. By combining program modules into one large program, the compiler has the opportunity to identify and consolidate re-entrant code, identify inconsistent variable declarations between modules, eliminate redundant code that may exist in more that one module and optimize all the stacks, registers, pointers and memories to exploit the unique advantages of the PSoC architecture.

Call Graph - A Clearer View

The intermediate code files generated by the OCG compiler are loaded into a *call graph* structure. Any library functions referenced by the program are also located and extracted.

Each calling convention contains a set of rules that defines which CPU registers are to be preserved across calls. *All* functions in every program module must adhere to the same calling conventions. Unfortunately, it is impossible to know at compile time which registers will and will not actually be used by a called function. In order to avoid potentially catastrophic consequences of not having a needed register, compilers frequently allocate more registers than are really necessary, thus wasting scarce PSoC SRAM resources.

Figure 3 – Call Graph

```
Machine type is 18F452

Call graph:

*_main size 0,6 offset 0
*     _dummy size 0,5 offset 6
*         _fcp size 2,12 offset 11
          awtoft size 0,0 offset 11
      _free size 0,2 offset 6
*_another_isr size 0,0 offset 25
*_my_isr size 0,6 offset 25
      lbtoft size 0,0 offset 31
*     _delay size 2,0 offset 31
```

In the case of the PSoC mixed-signal array, the same SRAM space is used for both software function stacks and data variables. If the programmer or compiler does not allocate sufficient SRAM space to accommodate the maximum depth of the dynamic stack, the stack can overflow into data variable space which could cause the program to crash. This is a problem with reentrant or recursive functions, which must either have dynamic stack space to store local variables or otherwise be managed to prevent them from overwriting existing data.

The OCG code generator uses the Call Graph to identify any functions that are called recursively or re-entrantly, such as those called from both main-line code and interrupt functions. It allocates dynamic stack space for storage of local variables to ensure that a re-entrant call of the function does not overwrite existing data. It also searches the call graph for any functions that are never called by the program and removes them.

The vast majority of functions are non-re-entrant and non-recursive, and may be implemented with a more predictable, static compiled stack. The compiler's OCG technology looks at all the program modules, identifies all non-re-entrant and non-recursive functions, and compiles an optimally sized function stack with exactly enough memory to accommodate each function's maximum depth. Because the call graph for all functions has already been determined, functions which are executed at different times can share the same SRAM space for their static compiled stack. This feature reduces stack space to the absolute minimum required and frees up more SRAM for data. A compiled static stack also reduces the likelihood of stack overflows that occur when a dynamic stack expands into the data variables' space in the SRAM.

At the end of this optimization, compiled stack space can be allocated before any machine code has been generated. OCG knows exactly how big the stack needs to be and where it is located before it generates any code.

Pointer Reference Graph Identifies Inconsistent Variable Declarations

Determining the memory space for each pointer is one of the most important features of the HI TECH C PRO compiler. Allocating too much memory to pointers wastes scarce SRAM resources on the mixed signal array.

When the stacks are optimized, the compiler builds reference graphs for all objects and pointers in the program. The OCG code generator has an algorithm that uses each instance of a variable having its address taken, plus each instance of an assignment of a pointer value to a pointer (either directly, via function return, function parameter passing, or indirectly via another pointer). It uses this information to build a data reference graph (Pointer Reference Graph) that identifies all objects that can possibly be referenced by each pointer. This information is used to determine exactly how much memory space each pointer will be required to access.

Variables and other objects that are used in multiple modules *must* have consistent definitions across all modules for the program to function properly. However, with programming teams dispersed across various facilities (that also may be on different continents), the rule is extremely difficult to enforce.

Programmers can mitigate the potential for error by having the linker check for incompatible variable re-declarations between modules. In the best case, the programmer can correct the variable inconsistencies in the C-code and re-compile it – a cumbersome, but effective approach. In the worst case, the linker doesn't have enough information to detect the inconsistency and the human error is compiled into the object code, adding to the debugging task.

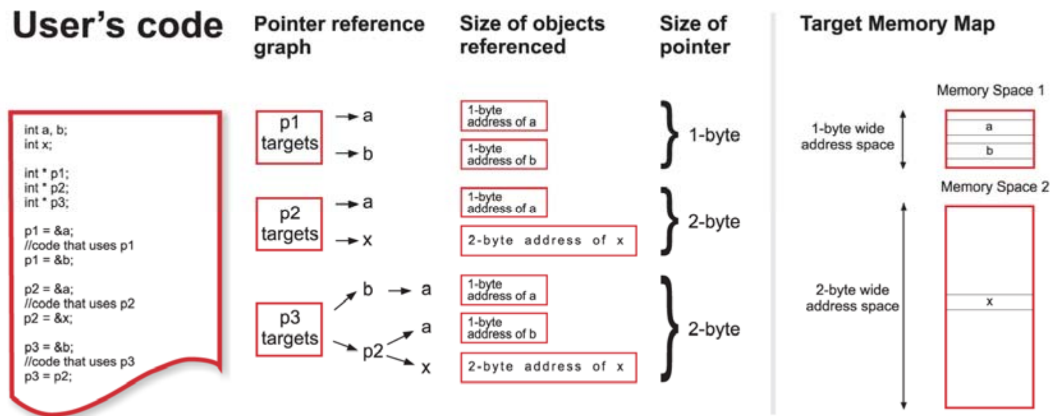
Since the OCG code generator sees all program modules at once, it is immediately able to identify variable or object declarations that are inconsistent among the different modules. If inconsistencies are found, the code generator issues an error message to the programmer that includes the names and locations of all variable names with inconsistencies so they can

[+] Feedback

be corrected prior to compilation. The code generator also flags variables that are never referenced and removes them. It also identifies functions that return values that are never used, so the code that prepares the return value can be eliminated.

The OCG code generator uses the data on all the pointers and variables in the pointer reference graph to determine the exact size of both the compiled and dynamic stacks and allocates memory to them. It also allocates the memory for global and static variables. Since the code generator has perfect information about pointer and variable usage, memory allocations are always optimized.

Figure 4 – Pointer Reference Graph



Optimizes Non-contiguous Memory Addresses

Standard C assumes a single linear address space, while in reality many embedded processors (including the PSoC mixed signal array) have complex, non-linear memory spaces, often with different address widths. For example, the PSoC mixed-signal array has a paged SRAM architecture, in which only 256 bytes of SRAM are addressable at any one time. Accessing any other memory page requires the page select register (PSR) to be reset. This is a cumbersome, cycle-intensive process that should be avoided if possible because each PSR reset takes three bytes of code and 12 cycles to execute. Thus, if data from a page that is already in use needs to be written to a different memory page, quite a bit of additional program code and clock cycles will be added to the program. The memory space to which variables are assigned suddenly becomes a very important determinant of both execution speed and code size.

This is particularly true for PSoC interrupts because the device automatically selects Page0 for interrupt routines. If the interrupt routine requires access to a variable in any page other than Page0, the PSR must be saved, the memory access mode changed, and the PSR loaded with the other page address. Afterwards the PSR must be restored to its state prior to the interrupt – for a total of 12-Bytes of extra program code and 50 clock cycles for every instance. In an extreme case, sub-optimal allocation of variables to the memory pages could easily double the program size *and* the number of cycles required to execute the routine.

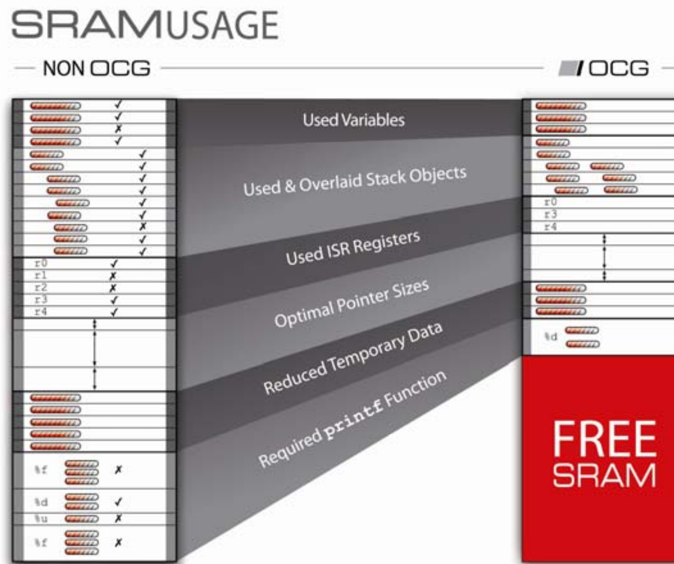
Conventional compilers are as likely as not to assign variables associated with interrupts to pages other than Page0 because they do not have enough information to do otherwise. A programmer who is aware of this issue can manually craft the code to ensure that all variables required for interrupt routines are stored in Page0. However, assembly code solutions compromise code portability.

Another means of solving this problem is to use the HI-TECH C PRO compiler. It's OCG code generator knows which variables will be used in interrupt routines and automatically assigns them to Page0.

The OCG code generator also sizes each pointer and encodes it in a way that is optimally efficient for the PSoC architecture. Each pointer is automatically assigned an optimized set of address spaces, without any intervention from the programmer.

[+] Feedback

Figure 5- SRAM- Use with Conventional and OCG Compilation



Bottom Up Code Generation

Once the pointers and variables are assigned memory space, machine code generation can begin. The OCG code generator begins at the bottom of the call graph, starting with those functions that do not call any other functions. Automatic in-lining of these functions may be performed if desired. In any case, the code can be generated without the constraints of rigid calling conventions. Code generation then proceeds up the call graph so that for each function the code generator knows exactly which functions are called by the current function. The code generator knows exactly which registers and other resources are available at each point. Calling conventions can be tailored to the register usage and argument type of a function, instead of following a blind set pattern.

Customized Library Functions

Most microcontrollers come with predefined C-code libraries of routinely performed functions. Two cases in point are the workhorse `sprintf()` and `printf()`functions used for formatting text strings or output. These functions have options for outputting text strings in many different formats and are enormously useful. However, when implemented in their entirety, they can occupy a code footprint of 5 KBytes or more.

Most programs only need a fraction of the available options, so the code can be reduced accordingly. The OCG code generator can analyze all the format strings in the program that are supplied to these functions, determine exactly the subset of format specifiers and modifiers required for the program, and create a customized version. The savings on code size can be immense. For example, the code for a minimal version of `sprintf()` that implements simple string copying can be as little as 20 or 30 Bytes, whereas a version providing real number formats with specific numbers of digits could occupy 5000 Bytes or more. No programmer input is required to benefit from this customization and optimization of C-library code.

Customized Runtime Startup Code

The C language requires uninitialized static and global variables to be cleared to zero on startup. Many newer embedded compilers provide canned startup code that performs this housekeeping function. However, canned startup code is often much larger than necessary for a given program. For example, if the program has no uninitialized global variables, there is no need to include code to clear them. OCG makes this information available to the code generator, which then creates custom runtime startup code. In a minimal case the startup code may be completely empty.

Legacy Code

Most software evolves over time**,** integrating existing hand crafted assembly code routines that have been developed and refined for earlier generations of the program. HI TECH C PRO for the PSOC Mixed Signal Array can combine the C program with externally supplied assembler and object modules. A pre-scan of these modules is done before code generation, and information is extracted that identifies any reserved memory areas, references to C functions, variables from assembler code,

[+] Feedback

and similar details. This information is passed to the code generator, allowing the legacy code to successfully integrate with the highly optimized generated code.

Summary

Unlike compilers that claim "global optimization", but really only optimize on individual program modules, Omniscient Code Generation looks at every module in the entire program and optimizes across all program modules. One obvious advantage of OCG is smaller, faster code. Code compiled for the mixed signal array using the OCG technology is about 50% smaller than code produced by conventional compilers. OCG can effectively double the amount of program code that can be stored in the device's on-chip flash, freeing it up for more code or additional dynamically configurable functions. A single C-language source file, compiled for Cypress' PSoC mixed signal array, was 51.3% smaller using the OCG compiler than that compiled by an existing third-party compiler.

Comparative code sizes (bytes) for ts057.c

Target Chip	Compiler	OCG	Code Size	∈%
Cypress PSoC	Conventional Compiler	no	11957	
	PRO for the PSoC Mixed Signal Array	yes	6129	-51.3%

Code size can be used as a proxy for performance. With less code to execute, the OCG compiler also improves execution speed.

OCG technology improves SRAM utilization by as much as 10% to 15%. Stack overflows are effectively eliminated.

Equally important, the HI-TECH C PRO compiler allows embedded C programs to be written *without* the use of architecture-specific extensions. The somewhat irregular architectures of embedded microcontrollers are often an awkward fit with the standard C language, frequently requiring substantial architecture-specific hand crafting to achieve efficient code. Omniscient Code Generation simplifies and streamlines the programmer's job by abstracting and hiding the underlying architecture, while simultaneously delivering reduced code size and increased execution speed. By performing an analysis of the whole program at compile time, the omniscient code generator can make optimal decisions about memory placement, pointer scoping, and stack allocation without any special directives or language extensions. The analysis is performed every time the program is recompiled, so it is always accurate and up-to-date.

Test Drive HI-TECH C PRO for the PSoC Mixed Signal Array

Designers who want to test the HI TECH C PRO compiler on existing or new C-code may down load a fully functional 45 day trial version, free of charge, at HI-TECH's website http://www.cypress.htsoft.com.

About the Author

Clyde Stubbs is the founder and CEO of HITECH Software. He graduated with honors in Computer Science from the University of Queensland, Queensland, Australia in 1982.

His university research in compiler technology led to the founding of HI-TECH Software in 1984. In 2006, he developed omniscient code generation (OCG) technology which generates object code based on a comprehensive analysis of all program modules, before compilation, allowing truly global optimization of stacks, registers, pointers and memories for much better code density that can be achieved by conventional compilation techniques.

Publications:

Whole-program Compiler Technology For Increased Code Density, Embedded Control Europe, June,2007

Using Whole-program Compilation to Improve MCU Code Density and Performance, embedded.com, March 15, 2007

About HI-TECH Software

HI-TECH Software (www.htsoft.com) is a world class provider of development tools for embedded systems, offering compilers and an Eclipse based IDE (HI-TIDE) for 8-, 16-, and 32-bit microcontroller and DSP chip architectures.

Founded by Clyde Stubbs, in 1984 in Brisbane, Australia, HI-TECH Software has a distribution center in Gilroy California, and an extensive network of distributors around the Globe.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com

© Cypress Semiconductor Corporation, 2007. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

[+] Feedback